Hierarchical-Alphabet Automata and Applications
Introduction

• Who am I?
  • PhD student at the AI Lab (Applied Research)
  • Under supervision of prof. Johan Loeckx
  • Research interests in applied AI and cybersecurity

• What is this guest lecture about?
  • Variants of finite state machines and applications
  • Our hierarchical extension on finite state machines
Outline of the lecture

• Preliminary knowledge:
  • Terminology + basics of automata theory
  • Tries, factor automata, and factor oracles

• Our research at the AI lab:
  • Hierarchical-alphabet automata (HAAs)
  • Hierarchical factor oracles (HFOs)

• Applications of our research:
  • Anomaly detection with the HFO
Outline of the lecture

• Preliminary knowledge:
  • Terminology + basics of automata theory
  • Tries, factor automata, and factor oracles

• Our research at the AI lab:
  • Hierarchical-alphabet automata (HAAs)
  • Hierarchical factor oracles (HFOs)

• Applications of our research:
  • Anomaly detection with the HFO
Terminology

- **Alphabet**: finite set of symbols
- **Words**: concatenations of zero or more symbols

- **Factors**: contiguous subsequences of a word
  - **Prefix**: a factor at the beginning of a word
  - **Suffix**: a factor at the end of a word

- **Language**: subset of the infinite set of possible words we can create using an alphabet!
Terminology: Examples

- **Alphabet:** \{ a, b, c, ..., 3, 4 \}
- **Words:** \{ "", "a", "abc", "cars", "3x4mpl3", ... \}

- **Factors:** *actor* is a factor of *factory*
  - **Prefix:** *fact* is a factor and prefix of *factory*
  - **Suffix:** *ory* is a factor and suffix of *factory*

- **Languages:** a*b*, the set of factors of a word, ...
Finite State Machines

- **Definition:** a five-tuple \((Q, \Sigma, \delta, q_0, F)\)
  - A finite set of states \(Q\)
  - An **alphabet** (or set of symbols) \(\Sigma\)
  - A transition function \(\delta\), returns a state \(q\) of \(Q\)
  - The **initial state** \(q_0\)
  - A set of **accepting states** \(F\)
Finite State Machines

- **Definition**: a five-tuple \((Q, \Sigma, \delta, q_0, F)\)
  - A finite set of states \(Q\)
  - An **alphabet** (or set of symbols) \(\Sigma\)
  - A transition function \(\delta\), returns a \(q\) of \(Q\)
  - The initial state \(q_0\)
  - A set of **accepting states** \(F\)
Finite State Machines

- **Definition:** a five-tuple \((Q, \Sigma, \delta, q_0, F)\)
  - A finite set of states \(Q\)
  - An *alphabet* (or set of symbols) \(\Sigma\)
  - A transition function \(\delta\), returns a \(q\) of \(Q\)
  - The initial state \(q_0\)
  - A set of *accepting states* \(F\)
Finite State Machines

- **Definition:** a five-tuple \((Q, \Sigma, \delta, q_0, F)\)
  - A finite set of states \(Q\)
  - An **alphabet** (or set of symbols) \(\Sigma\)
  - A **transition function** \(\delta\), returns a \(q\) of \(Q\)
  - The initial state \(q_0\)
  - A set of **accepting states** \(F\)
Finite State Machines

• **Definition:** a five-tuple \((Q, \Sigma, \delta, q_0, F)\)
  - A finite set of states \(Q\)
  - An *alphabet* (or set of symbols) \(\Sigma\)
  - A transition function \(\delta\), returns a \(q\) of \(Q\)
  - The *initial state* \(q_0\)
  - A set of *accepting states* \(F\)
Finite State Machines

- **Definition:** a five-tuple \((Q, \Sigma, \delta, q_0, F)\)
  - A finite set of states \(Q\)
  - An **alphabet** (or set of symbols) \(\Sigma\)
  - A transition function \(\delta\), returns a \(q\) of \(Q\)
  - The **initial state** \(q_0\)
  - A set of **accepting states** \(F\)
Finite State Machines

- Acceptance and rejection of words:
  - A finite state machine recognizes a language:
    - It accepts words part of that language
    - It rejects words not part of the language
  - Start at the initial state $q_0$ + first symbol of your input word
  - Repeatedly follow $\delta$ with current state + symbol of word:
    - Accepting state after full word? Word is accepted
    - Normal state or no transition? Word is rejected
Applications of FSMs

• Anomaly detection on system call sequences

Tries

- **Definition:**
  - Rooted tree associated with a set of words

- **Paths from root to leaf**
  represents words from its set

- **Example:**
  - Trie for the set \{ three, tree, trie \}
Tries

- **Definition:**
  - Rooted tree associated with a set of words

- **Paths from root to leaf** represents words from its set

- **Example:**
  - Trie for the set \{ three, tree, trie \}
Tries

• Definition:
  • Rooted tree associated with a set of words

• Paths from root to leaf represents words from its set

• Example:
  • Trie for the set { three, tree, trie }
Tries

- **Definition:**
  - Rooted tree associated with a set of words

- **Paths from root to leaf**
  represents words from its set

- **Example:**
  - Trie for the set \{ three, tree, trie \}
**Tries**

- **Definition:**
  - Rooted tree associated with a set of words

- **Paths from root to leaf** represents words from its set

- **Example:**
  - The word *tre* is not in { *three, tree, trie* }
Construction Algorithm of Tries

{trie, tree}
Construction Algorithm of Tries

{trie, tree}
Construction Algorithm of Tries

\{trie, tree\}
Construction Algorithm of Tries

{trie, tree}
Construction Algorithm of Tries
Construction Algorithm of Tries

\{\text{trie, tree}\}
Construction Algorithm of Tries

\begin{center}
\begin{tikzpicture}
  \node[shape=circle,draw=black] (1) at (0,0) {t};
  \node[shape=circle,draw=black] (2) at (1,0) {r};
  \node[shape=circle,draw=black] (3) at (2,1) {i};
  \node[shape=circle,draw=black] (4) at (2,2) {e};
  \node[shape=circle,draw=black] (5) at (1,-1) {e};
  \node[shape=circle,draw=black] (6) at (0,-2) {e};
  \path[->] (1) edge node {} (2);
  \path[->] (2) edge node {} (3);
  \path[->] (3) edge node {} (4);
  \path[->] (4) edge node {} (5);
  \path[->] (5) edge node {} (6);
\end{tikzpicture}
\end{center}

\{trie, tree\}
Construction Algorithm of Tries
Construction Algorithm of Tries

{trie, tree}
Construction Algorithm of Tries
Construction Algorithm of Tries

\{trie, tree\}
Applications of Tries

- Efficient data structure, storing words based on prefixes
  - Eliminates duplicate prefixes, allows for efficient search
Interactive Demonstration of Tries
Factor Automata

- **Definition:**
  - Factor automaton of a word \( x \) is the *minimal* deterministic automaton that recognizes the factors of \( x \)

![Diagram of a minimal deterministic automaton recognizing factors of aabbabb.](image)

*Fig. 7.11. Minimal deterministic automaton recognizing the factors of aabbabb.*
Factor Oracles

• **Definition:**
  - Acyclic automaton built on a (set of) word(s), recognizes **at least** all factors of this (set of) word(s)

• **Applications:**
  - Intended for pattern matching
  - Computation of repeat factors
  - Modelling for music improvisation

Factor Automata vs Factor Oracles

- **Factor automaton:**
  - Deterministic automaton that recognizes the factors of $x$

- **Factor oracle:**
  - Acyclic automaton that recognizes **at least** all factors of $x$

Fig. 7.11. Minimal deterministic automaton recognizing the factors of aabbabb.
Factor Automata vs Factor Oracles

- **Factor oracle:**
  - Has an *online algorithm*
  - Constructed in *linear time and space*
  - Memory-efficient improvement over FAs

- **Example:**
  - Factor oracle for \{three, trie, tree\}
Factor Oracles

- **States and transitions:**
  - Has exactly $m + 1$ states
  - Between $m$ and $2m - 1$ transitions

- **Examples:**
  - FO for the word "factor"
  - FO for the word "bubble"
  - FO for the word "eeeeeee"
Factor Oracles

• **States and transitions:**
  • Has exactly $m + 1$ states
  • Between $m$ and $2m - 1$ transitions

• **Example: "factor"**
  • Word has length 6
  • Has 7 ($= 6 + 1$) states
  • Has 11 ($= 2 \times 6 - 1$) transitions
  • **Why?** No repeat symbols = *worst-case*
Factor Oracles

- **States and transitions:**
  - Has exactly $m + 1$ states
  - Between $m$ and $2m - 1$ transitions

- **Example: "eeeeee"**
  - Word has length 6
  - Has 7 (= 6 + 1) states
  - Has 6 transitions
  - Why? Every symbol is equal = **best-case**
Factor Oracles

- **Accepts at least all factors:**
  - FO for \{three, trie, tree\}

- **Example:**
  - Accepts "hre", which is a factor of \texttt{three}
Factor Oracles

- Accepts at least all factors:
  - FO built on \{three, trie, tree\}
- Example:
  - Accepts "ree", which is a factor of both three and tree
Factor Oracles

• **Accepts at least all factors:**
  - FO built on \{three, trie, tree\}

• **Example:**
  - Rejects "trh", because it is not a factor of three, trie, or tree
Construction Algorithm of FOs

(a) The trie for $P = \{\text{trie, tree, three}\}$, taken from Fig. 2.10. In the first step of the algorithm, we generate the trie which we use as a starting point to generate the factor oracle.

(b) In this step of the construction algorithm, a new transition from the initial state to state 2 is made, labelled with $r$. There are now 11 transitions in total.
Construction Algorithm of FOs

(c) In this step of the construction algorithm, a new transition from the initial state to state 7 is made, labelled with $h$. There are now 12 transitions in total.

(d) In this step of the construction algorithm, a new transition from the initial state to state 3 is made, labelled with $i$. There are now 13 transitions in total.
(e) In this step of the construction algorithm, a new transition from the initial state to state 5 is made, labelled with $e$. There are now 14 transitions in total.

(f) In the final step of the construction algorithm, all states are marked as final states. The algorithm transformed the original trie to the factor oracle of $P = \{\text{trie}, \text{tree}, \text{three}\}$. 
Interactive Demonstration of FOs
Applications of FOs

• **Used for pattern matching:**
  • Backwards Oracle Matching (BOM)
  • Set Backwards Oracle Matching (SBOM)
Set Backward Oracle Matching

(a) We attempted to input \_ in the factor oracle, but the oracle rejects directly. We shift the window after the \_ and continue our search.
Set Backward Oracle Matching

two or three trees

(b) We successfully input $t$ in the factor oracle, but the oracle fails on $\_$. We shift the window after the $\_$ and continue our search.
(e) We successfully input $e$, $r$, $h$, and $t$ in the oracle. We reach accepting state 10, which is associated with the word *three*. We check for an occurrence of *three*, and shift the window by 1.
We successfully input e, e, and r in the oracle, but the oracle fail on the character h. We shift the window after the h and continue our search.
two or three trees

(e) We attempted to input \( \Diamond \) in the factor oracle, but the oracle rejects directly. We shift the window after the \( \Diamond \) and continue our search.
Set Backward Oracle Matching

(f) We input $e, e, r,$ and $t$ in the oracle. We reach accepting state 7, which is associated with the word *tree*. Our window contains *tree*.
Applications of FSMs (2)

• Anomaly detection on system call sequences:
  • Could we use factor oracles for this?

• Problems:
  • Explosion in size
  • Narrowly-defined vs. broad behavior

Outline of the lecture

• Preliminary knowledge:
  • Terminology + basics of automata theory
  • Tries, factor automata, and factor oracles

• Our research at the AI lab:
  • Hierarchical-alphabet automata (HAAs)
  • Hierarchical factor oracles (HFOs)

• Applications of our research:
  • Anomaly detection with the HFO
Research Goals

• Solving previously-mentioned issues:
  • Capturing broad, complex behavior with compact FSMs

• Possible solution:
  • Exploiting *hierarchical relationships* of symbols

• How?
  • Words to *hierarchical words* using hierarchical relationships
  • Factors to *hierarchical factors*
  • Finite state machines to *alphabet-hierarchical automata*
Hierarchical Words

- **Alphabet**: finite set of symbols
- **Hierarchical words**: concatenations of symbols in the form of a rooted tree
- **Hierarchical factors**: factors of hierarchically connected words for every level of the hierarchy
Hierarchical Words

- **Alphabet**: finite set of symbols
- **Hierarchical words**: concatenations of symbols in the form of a rooted tree
- **Hierarchical factors**: factors of hierarchically connected words for every level of the hierarchy
Hierarchical Words

• Where is hierarchy relevant?
  • **Language**: words, sentences, paragraphs
  • **Music**: individual notes, chords
  • **Action**: actions into sequence of sub-actions

• Hierarchy in cybersecurity?
  • **Parent-child relationships between processes**: one process can spawn others
Hierarchical-Alphabet Automata

• Intuitively:
  • Hierarchical variant of regular FSMs
  • Has a super-transition function $\Delta$ that maps symbols of its alphabet in one level to (at most) one other HAA
Hierarchical Factor Oracles

- Hierarchical-alphabet variant of regular (flat) factor oracles:
  - Built using a (set of) hierarchical word(s)
  - Accepts hierarchical factors of its set
Hierarchical Factor Oracles

- Hierarchical-alphabet variant of regular (flat) factor oracles:
  - Built using a (set of) hierarchical word(s)
  - Accepts hierarchical factors of its set
Hierarchical Factor Oracles

- Hierarchical-alphabet variant of regular (flat) factor oracles:
  - Built using a (set of) hierarchical word(s)
  - Accepts hierarchical factors of its set
Construction Algorithm for HFOs

- Hierarchical oracle for a set of hierarchical sequences:
  - Perform a **breadth-first traversal** on each hierarchical sequence
  - During the traversal, build a list of regular sequences
  - After traversal: build regular FOs using regular sequences
  - Create super-transitions from oracle to oracle
Construction Algorithm for HFOs

- Hierarchical oracle for a set of hierarchical sequences:
  - Perform a **breadth-first traversal** on each hierarchical sequence
  - During the traversal, build a list of regular sequences
  - After traversal: build regular FOs using regular sequences
  - Create super-transitions from oracle to oracle
Construction Algorithm for HFOs

- Hierarchical oracle for a set of hierarchical sequences:
  - Perform a breadth-first traversal on each hierarchical sequence
  - During the traversal, build a list of regular sequences
  - After traversal: build regular FOs using regular sequences
  - Create super-transitions from oracle to oracle
Construction Algorithm for HFOs

- Hierarchical oracle for a set of hierarchical sequences:
  - Perform a **breadth-first traversal** on each hierarchical sequence
  - During the traversal, build a list of regular sequences
  - After traversal: build regular FOs using regular sequences
  - Create super-transitions from oracle to oracle
Accepting or rejecting input

Accepting with
Accepting or rejecting input

Stack = []
Current = None
Accepting or rejecting input

Stack = [$\beta$]
Current = None
Accepting or rejecting input

Stack = [α, β]
Current = None
Accepting or rejecting input

Stack = [$\beta$]
Current = $\alpha$
Accepting or rejecting input

Stack = [b, β]
Current = α
Accepting or rejecting input

Stack = \([a, b, \beta]\)
Current = \(\alpha\)
Accepting or rejecting input

Stack = [b, β]
Current = a
Accepting or rejecting input

Stack = [β]
Current = b
Accepting or rejecting input

Stack = []
Current = \( \beta \)
Accepting or rejecting input

Stack = [c]
Current = β
Accepting or rejecting input

Stack = []
Current = c
Accepting or rejecting input

Stack = []
Current = None
Accepting or rejecting input

accepted by
Interactive Demonstration of HFOs
Outline of the lecture

• Preliminary knowledge:
  • Terminology + basics of automata theory
  • Tries, factor automata, and factor oracles

• Our research at the AI lab:
  • Hierarchical-alphabet automata (HAAs)
  • Hierarchical factor oracles (HFOs)

• Applications of our research:
  • Anomaly detection with the HFO
Anomaly Detection

• **What is it?**
  - Finding patterns in data that do not conform to expected behavior
  - Relevant in lots of domains and applications, and real-life relevance!

• **(Time) series anomaly detection:**
  - Detecting spikes, drops, out-of-place patterns, unexpected trends, …
HPM-based Anomaly Detection

- **Main focus:** anomaly detection in production environments
  - Needs to be **lightweight** due to constraints in resources
  - We should never forget **rarely-occurring events**
  - We should be able to adapt to **new behavior**
  - It should be **general** enough to work on different time series
  - It should **recognize non-strictly periodic events**
  - Should have **a minimum amount of false positives**

HPM-based Anomaly Detection

- **Key idea**: extract hierarchical fingerprints from time series
  - Store all seen fingerprints in a tree structure
  - New fingerprints are marked as possible anomalies

HPM-based Anomaly Detection

Graphs showing data for different depths:
- depth = 0
- depth = 1
- depth = 2
- depth = 3

Scale for x-axis: 0 to 150
Scale for y-axis: -1.0 to 1.0

Artificial Intelligence Research Group
HPM-based Anomaly Detection
HPM-based Anomaly Detection
HPM-based Anomaly Detection
Interactive Demonstration of HPM
HPM-based Anomaly Detection

- Disadvantages of basic algorithm:
  - Everything is anomalous if you go fine-grained enough
  - No temporal anomalies: only new patterns are possible anomalies
  - Lack of context: have we seen a factor of fingerprints before?
Extending HPM with HFOs

• Possible approach:

1. Segmenting: extract subsequences using a sliding window

2. Discretizing: extract fingerprints from above-obtained subsequences

3. Learning: represent consecutive fingerprints as hierarchical words (and HFO)

4. Anomaly detection: calculate score based on largest accepted HFO input
Extending HPM with HFOs

- Segmenting:

```plaintext
requests
2022-07-31 22:00:00+00:00 29309
2022-07-31 23:00:00+00:00 29977
2022-08-01 00:00:00+00:00 29748
2022-08-01 01:00:00+00:00 28414
2022-08-01 02:00:00+00:00 27871
...  ...
2022-10-14 17:00:00+00:00 41931
2022-10-14 18:00:00+00:00 41215
2022-10-14 19:00:00+00:00 38529
2022-10-14 20:00:00+00:00 37104
2022-10-14 21:00:00+00:00 34412
[[29309 29977 29748 ... 40237 36560 34145]
[29977 29748 28414 ... 36560 34145 31998]
[29748 28414 27871 ... 34145 31998 31041]
...
[37564 34118 29764 ... 41931 41215 38529]
[34118 29764 29600 ... 41215 38529 37104]
[29764 29600 29485 ... 38529 37104 34412]]
```
Extending HPM with HFOs

• Discretizing:

\[
\begin{array}{c}
\{29309 \ 29977 \ 29748 \ \ldots \ 40237 \ 36560 \ 34145\} \\
\{29977 \ 29748 \ 28414 \ \ldots \ 36560 \ 34145 \ 31998\} \\
\{29748 \ 28414 \ 27871 \ \ldots \ 34145 \ 31998 \ 31041\} \\
\ldots \\
\{37564 \ 34118 \ 29764 \ \ldots \ 41931 \ 41215 \ 38529\} \\
\{34118 \ 29764 \ 29600 \ \ldots \ 41215 \ 38529 \ 37104\} \\
\{29764 \ 29600 \ 29485 \ \ldots \ 38529 \ 37104 \ 34412\}
\end{array}
\]

\[
\begin{array}{c}
\{\text{Node(depth, slope, inter, \ldots)}, \ \text{Node(\ldots)}, \ \ldots\} \\
\{\text{Node(depth, slope, inter, \ldots)}, \ \text{Node(\ldots)}, \ \ldots\} \\
\{\text{Node(depth, slope, inter, \ldots)}, \ \text{Node(\ldots)}, \ \ldots\} \\
\ldots \\
\{\text{Node(depth, slope, inter, \ldots)}, \ \text{Node(\ldots)}, \ \ldots\} \\
\{\text{Node(depth, slope, inter, \ldots)}, \ \text{Node(\ldots)}, \ \ldots\} \\
\{\text{Node(depth, slope, inter, \ldots)}, \ \text{Node(\ldots)}, \ \ldots\}
\end{array}
\]
Extending HPM with HFOs

- Learning:

\[
[[\text{Node}(\text{depth}, \text{slope}, \text{inter}), \text{Node}(\ldots), \ldots]  \\
[\text{Node}(\text{depth}, \text{slope}, \text{inter}), \text{Node}(\ldots), \ldots]  \\
[\text{Node}(\text{depth}, \text{slope}, \text{inter}), \text{Node}(\ldots), \ldots]  \\
\ldots  \\
[\text{Node}(\text{depth}, \text{slope}, \text{inter}), \text{Node}(\ldots), \ldots]  \\
[\text{Node}(\text{depth}, \text{slope}, \text{inter}), \text{Node}(\ldots), \ldots]  \\
[\text{Node}(\text{depth}, \text{slope}, \text{inter}), \text{Node}(\ldots), \ldots]]
\]
Extending HPM with HFOs

- Learning:
Extending HPM with HFOs

• Advantages over HPM:
  • Not everything is anomalous: search finds largest accepting factor
  • Temporal anomalies: old patterns are possibly anomalous
  • Lack of context: checks for factors of fingerprints

• Advantages over HPM (extended with other state methods):
  • Compactness: exploiting hierarchy of discretization for compactness
  • Hierarchy: takes hierarchy of discretization into account
Interactive Demonstration of Algorithm